TERA

Multithreaded Architecture (MTA)

Parallel Computing : State and Hurdles

- A truly general purpose parallel computer is yet to materialize
- There are wide variety of parallel computers
- Each has different network interconnect, cache arrangement, and different programming languages (PVM, MPI, HPF, parallel libraries etc.)
- With current parallel computers there is a never ending battle to match computation to architecture

TERA Approach

- Reduce programmer’s effort and cost
- Build processors, memories, and networks to facilitate parallel programming, rather than build systems out of commodity PC parts

TERA Architecture

- TERA MTA is a large parallel machine built from proprietary multithreaded processors
- A large flat uniform access shared memory
- High bandwidth connections between processor and memory units, allowing every processor to receive a data word every clock tick
- Abundant lightweight synchronization via full/empty bit associated with every word of memory
- Context switching between threads at every clock ticks

TERA Architecture (cont..)

- Each processor has hardware support for many threads and can effectively use high level of parallelism on a single processor
- Processor switches to a ready thread at each clock tick and thus is able to stay busy in the face of all sorts of latencies
- Ability to tolerate memory latency is the primary reason why the MTA is able to provide high utilization and scalability
- Performance is independent of data locality

Streams

- Each MTA processor has 128 “streams” each of which is hardware (including 32 registers and a program counter) that is devoted to running single thread of control
- The processor executes instructions from streams, that are not blocked, in a fair round robin fashion
- A stream can issue an instruction every 21 cycles (the length of the instruction pipeline) so at least 21 ready threads are required to keep a processor fully busy
- The processor makes a context switch on each cycle, choosing the next instruction from one of the streams that is ready to execute

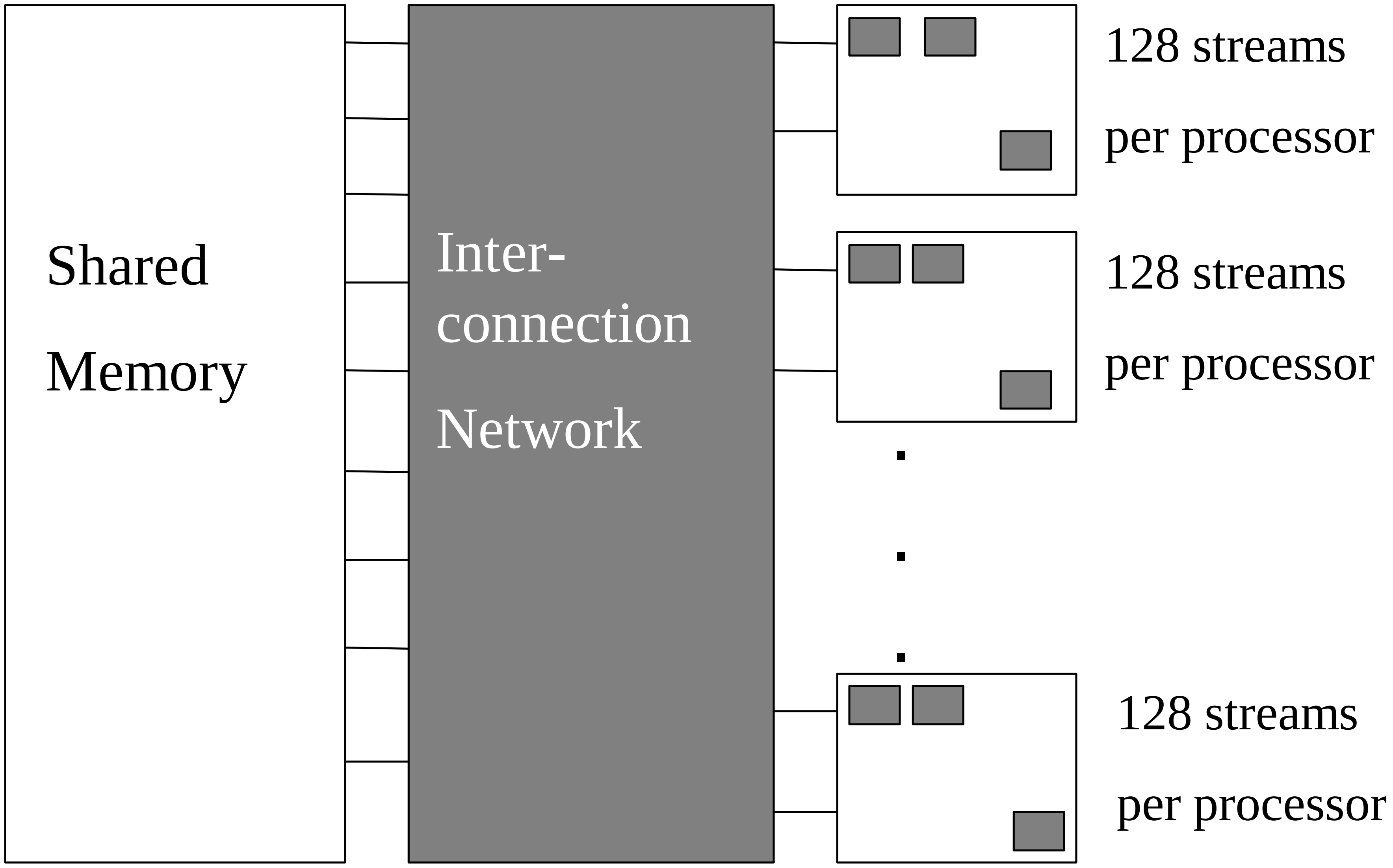
Threads

- Threads are software entity
- When a thread is assigned to a stream, its instructions are executed one at a time requiring 21 cycles per instruction
- Each instruction includes a "lookahead" number (between 0 and 7) that designates how many additional instructions can be executed before the result of the memory operation is needed
- Since memory operations require 5 or 6 times the 21-cycle execution speed, fast thread execution requires that the compiler be able to schedule memory operations ahead of when their results are needed

TERA Memory

- Flat shared memory : all data accessible with equal ease - No locality - No cache - No mapping - No stride sensitivity
- Latency to memory of ~140 cycles is tolerated by having more that 21 threads, each with lookahead or performing non-memory operations
- All memory words are 64 bits with 4 additional access state bits
- Memory addresses are randomly scattered across memory banks
- Extremely fine grained synchronization by full/empty bits on each word

A View of the TERA Multiprocessor



Status of TERA MTA at SDSC

- First TERA MTA processor was delivered to SDSC in November, 1997 with a single 145 MHz processor (< 1/2 final speed)
- Currently a 4 processor system with 260 MHz (final clock speed will be 300 MHz)
- TERA's unix OS called MTX is available; newer versions being released
- Performance measurement on the TERA at SDSC: NAS parallel benchmark codes, molecular dynamics code (AMBER), finite element code (LS-DYNA3D), fluid code (LCPFCT), battle field simulation code etc.

T90/MTA Hardware Comparison

- CRAY T90:
  440 MHz
  8 128-element V registers/cpu
  Dual vector pipes into FUs
  Pipelines Add & Mult units
  Can execute 4 flops/cy
  (commonly 2)
  Flat shared memory
  SRAM, high BW, low latency
  Can issue 2 loads + 1 store /cy
- Peak1.76Gflops/CPU
  Practical peak of 1 Gflops
  Observed 400-800 Mflops in
  “good” user code

- TERA MTA-1
  260 MHz (300+ MHz final)
  128 streams (h/w for threads)/cpu
  Effective depth of pipeline is 21
  Additional FMA unit
  Can execute 3 flops/cy
  (commonly 2)
  Flat shared memory
  SDRAM, moderate latency, BW
  Can issue 1 memory ref/cy
- Peak 0.9+ Gflops/CPU
  Practical peak of 600 Mflops
  Tera expects sustained 30-60%of
  peak in “good” user code

Parallelism on the TERA

- Multiple levels of parallelism
  - Execute outer loops independently, concurrently across multiple processors
  - On each processor, across multiple streams
  - Within each stream, several memory references may be outstanding while other instructions are executing
  - Within each instruction : lookahead, 3 operations

Parallelism on the TERA (cont..)

- Parallelizing loops
- Conditions for parallelizing :
  - First the loop must be an inductive loop so that it is possible to determine how many iterations will be executed before the loop begins
  - Secondly the loop must be of a form that the compiler can handle i.e. a parallel loop where each iteration can be executed independent of others or linear recurrence

Parallelism on the TERA (cont..)

– Compiler automatically detects and manages loop parallelism, including synchronization

```
    do I = 1, N
       E(I) = G(I) + A(I)
     end do
```

– User can influence compiler via compiler directives

```
C$TERA ASSERT PARALLEL
     do I = 1, N
      E(IDX(I)) = E(IDX(J)) + foo(I)
     end do
```

Parallelism on the TERA (cont..)

- C$TERA ASSERT PARALLEL can be quite useful
- Needs to be used with care (compiler does not necessarily perform the same transformation for asserted loops that it does for others)

```
C$TERA ASSERT PARALLEL
    do I = 1, N
         K = KEY(I)
         ICONT(K) = ICONT(K) + 1
    end do
```

(continued in next slide…..)

Parallelism on the TERA (cont..)

- To ensure correct execution, synchronization must be provided explicitly

```
C$TERA ASSERT PARALLEL
   do I = 1, N
        K = KEY(I)
        call INT_FETCH_ADD(ICOUNT(K),1)
   end do
```

- Forcing parallelization is not always possible, if a loop is not inductive compiler cannot parallelize regardless of any assertions

Behavior of full/empty bits

- A synchronized write into a variable succeeds only if it is empty, when the write completes, the location is set full
- A synchronized read from a variable succeeds only if it is full, when the read completes, the location is set empty
- A thread attempting a synchronized write (read) into a full (empty) location will be suspended by hardware and will resume only when that location becomes empty (full)
- Many ways to declare a sync variable

NAS 2.3 Serial Benchmark results

Benchmark	CRAY T90 (440 Mhz)	TERA (260 Mhz)
CG	171 Mflops(1PE)	171 Mflops(1PE)
	378Mflops(4PE)	596 Mflops(4PE)
FT	774Mflops(1PE)	187 Mflops(1PE)
	2620Mflops(4PE)	701Mflops(4PE)
MG	576Mflops(1PE)	184 Mflops(1PE)
	2099Mflops(4PE)	702Mflops(4PE)
EP	6.80 Mops(1PE)	7.5 Mops(1PE)
	27.3Mops(4PE)	29.5Mops(4PE)
IS	42 Mops(1PE)	110Mops(1PE)
	164 Mops(4PE)	366Mops(4PE)

So why is TERA expected to be revolutionary?

- Ease of parallel programming : no need to worry about memory arrangement, data layout, data locality
- TERA hides (does not reduce!) the main bottleneck of current fast processors i.e. memory latency by multithreading
- No code change required going from single PE to multiple PEs
- Potential for seamless scaling to multiple processors
- TERA web at : http://www.tera.com